UNITÉ DE RECHERCHE
INRIA-ROCQUENCOURT

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P. 105
78153 Le Chesnay Cedex
France
Tél.:(1)39 63 55 11

Rapports de Recherche

N°968

Programme 2

Calcul symbolique, Programmation et Génie logiciel

ALAS : UN SYSTÈME D'ANALYSE ALGÉBRIQUE

Paul ZIMMERMANN

Janvier 1989

Alas: An Algebraic Analyzer System

Paul Zimmermann [1]

Abstract. $\Lambda\Upsilon\Omega$ (INRIA Research Report 876, July 1988) is a system designed to perform automatic average case analysis of well-defined classes of algorithms operating over "decomposable" data structures. It consists of an 'Algebraic Analyzer' System that compiles algorithms specifications into generating functions of average costs, and an 'Analytic Analyzer' System that extracts asymptotic informations on coefficients of generating functions. This report describes the Algebraic Analyzer (Alas). The Analytic Analyzer is described in a companion report by Bruno Salvy.

Recent research work in combinatorial analysis has established systematic correspondences between structural definitions and counting generating functions. This report extends these correspondences to program schemes. Alas is a system written in Caml, which enables to specify an algorithm and its companion data structures. When this algorithm belongs to a well-defined class, Alas computes associated generating functions. These generating functions can then be passed to the Analytic Analyzer that will give an asymptotic expansion of the average cost.

Alas : un système d'analyse algébrique

Résumé. $\Lambda\Upsilon\Omega$ (Rapport de recherche INRIA 876) est un système conçu pour réaliser automatiquement l'analyse en moyenne de classes bien définies d'algorithmes opérant sur des structures de données "décomposables". Il consiste en un 'Analyseur Algébrique' qui compile des spécifications d'algorithmes en séries génératrices de coût moyen ainsi qu'en un 'Analyseur Analytique' qui extrait des informations asymptotiques sur les coefficients de séries génératrices. Ce rapport décrit l'Analyseur Algébrique (Alas). L'Analyseur Analytique est décrit dans un autre rapport, fait par Bruno Salvy.

De récents travaux d'analyse combinatoire ont établi des correspondances systématiques entre les définitions de structures et les séries génératrices de dénombrement. L'objet de ce rapport est de prolonger ces correspondances à des schémas de programmes. Alas est un système écrit en Caml qui permet de spécifier un algorithme et ses structures de données. Lorsque cet algorithme fait partie d'une classe bien définie, Alas détermine les séries génératrices associées. Celles-ci pourront ensuite être passées à l'Analyseur Analytique qui donnera un développement asymptotique du coût moyen.

[1] Projet Algorithmes, Batiment 10, INRIA Rocquencourt, 78153 Le Chesnay Cédex

Table des matières

Liste des figures

Notations

$\|a\|$	taille de a
$f(z)$	série génératrice
f_n ou $[z^n]f$	coefficient de z^n de f
$\tau P(z)$	descripteur de complexité
τP_n	coefficient de z^n de τP
$\mathcal{A}$	ensemble d'objets
$\mathcal{A}_n$	sous-ensemble des objets de $\mathcal{A}$ de taille n
$\mathcal{A}^*$	$\epsilon \cup \mathcal{A} \cup \mathcal{A}^2 \cup \mathcal{A}^3 \cup \ldots$
$2^{\mathcal{A}}$	ensemble des parties (finies) de $\mathcal{A}$
$\mathcal{M}(\mathcal{A})$	multi-ensemble d'éléments de $\mathcal{A}$
$\mathcal{C}(\mathcal{A})$	ensemble des cycles d'éléments de $\mathcal{A}$
$\mathcal{UC}(\mathcal{A})$	cycles non orientés d'éléments de $\mathcal{A}$
$(a_1, \ldots, a_k)$	séquence
$\{a_1, \ldots, a_k\}$	ensemble ou multi-ensemble
$[a_1, \ldots, a_k]$	cycle
$\Phi(a)$	$\frac{a(z)}{1} - \frac{a(z^2)}{2} + \frac{a(z^3)}{3} - \cdots$
$\Psi(a)$	$\frac{a(z)}{1} + \frac{a(z^2)}{2} + \frac{a(z^3)}{3} + \cdots$

Chapitre 1

Introduction

1.1 Comment lire ce rapport ?

Le lecteur pressé peut se référer directement aux exemples du chapitre 10. Le chapitre 2 montre comment se servir des séries génératrices pour compter des objets. Le lecteur familiarisé avec cet outil peut passer au chapitre 3, qui présente les notions de descripteur de complexité et de schéma de programmation admissible. Un certain nombre de règles donnant l'opérateur correspondant à un schéma admissible y sont énumérées. Le chapitre 4 montre comment appliquer ces règles à l'analyse d'un algorithme.

Le chapitre 5 montre quelles sont les structures de données utilisées, l'interface avec Maple est l'objet du chapitre 6. Quant au chapitre 7, il indique comment sont résolues les équations produites par le programme Caml, en vue de l'analyse asymptotique.

Pour décrire un algorithme dans le langage Adl, il faut se référer au chapitre 8. Le lecteur trouvera au chapitre 9 comment lancer l'analyse de son programme avec $\Lambda\Upsilon\Omega$, et les options possibles. De nombreux exemples au chapitre 10 montrent comment résoudre avec $\Lambda\Upsilon\Omega$ un problème donné.

On pourra voir les fichiers Caml constituant Alas en consultant l'appendice A, le fichier Maple "Solve" se trouve dans l'appendice B et l'appendice C est le texte intégral d'une session d'analyse.

1.2 Qu'est-ce qu'un système d'analyse d'algorithmes ?

C'est un programme qui demande une définition de procédure:

```
function Derive(e:expression):expression;
begin
      {instructions}
end;
```

et qui fournit des informations sur le coût moyen de cette procédure:

```
Le cou^t moyen de Derive sur les expressions de taille n est:

                                 1/2    1/2  3/2
                        (126 + 6   ) Pi    n
                1/2 ------------------------------ + O(n)
                                1/2 3/2  3/4   1/2
                     (- 1 + 2 6   )    6    23
```

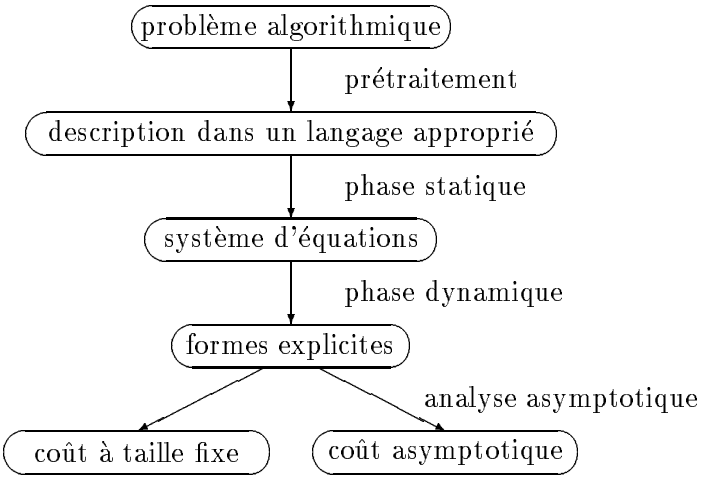
Un tel système permettrait de déterminer l'ordre de grandeur de complexité d'un algorithme (n, $n \log n$, n^2, n^3, c^n, n^n), de comparer deux algorithmes de même ordre de grandeur (en comparant les constantes multiplicatives). Ainsi, il serait possible de chiffrer rapidement l'influence de modifications d'un programme (par exemple une permutation de tests).

Le système $\Lambda\Upsilon\Omega$, né début 1988, permet déjà d'analyser un certain nombre de procédures, dont la fonction `Derive` (cf 10.2.8). Nous nous restreignons à des coûts *additifs* tels que le temps, le nombre de passages par tel endroit, ...(ceci exclut entre autres l'analyse en mémoire). L'analyse dépend de la procédure considérée, mais aussi du *modèle de complexité* (le temps que nécessite chaque instruction). Ainsi, il faut pouvoir indiquer par exemple qu'une affectation prend 2 unités de temps, un test 5 unités, un appel de procédure 11 unités. D'autre part, il faut pouvoir préciser le *modèle statistique des données*, c'est-à-dire quelles sont les données possibles et quelle est leur probabilité. Nous nous limiterons à un modèle statistique équiprobable.

1.3 Principes généraux

L'analyse de complexité d'un algorithme peut se faire en 4 phases:

1. description dans un langage approprié (prétraitement)
2. traduction en un système d'équations (phase statique)
3. résolution par un logiciel de calcul formel (phase dynamique)
4. calcul des valeurs de complexité recherchées (analyse asymptotique)



1.4 Choix d'un domaine

La phase statique fait correspondre à chaque procédure un objet mathématique, contenant des informations diverses, dont au minimum le coût moyen. Le *domaine* est l'espace de ces objets. Les domaines employés actuellement sont, à ma connaissance:

1. les fonctions de distribution de coût
2. les séries génératrices de dénombrement et de coût

La fonction de distribution d'un programme fait correspondre à chaque distribution des données la distribution des résultats. Les séries génératrices, quant à elles, contiennent le coût total du programme pour chaque taille des données. Chaque domaine permet d'analyser une certaine classe d'algorithmes. Le langage de description est choisi de telle sorte que ces algorithmes puissent être décrits facilement.

L'équipe de J. Cohen travaille depuis plusieurs années dans le domaine des fonctions de distribution. Dans un article récent [Cohen 88], Jacques Cohen montre d'abord qu'il sait analyser automatiquement des programmes écrits dans un langage simple (SL), par exemple le problème de la ruine du joueur (*The Gambler's Ruin*). Ce langage autorise les affectations, les boucles simples et les conditionnelles, mais pas les procédures. Ensuite, afin d'analyser des programmes plus compliqués, il définit un langage fonctionnel proche de Lisp, nommé FP. A une fonction FP est associée une fonction de distribution. Ainsi, à chaque endroit du programme, la distribution de chaque variable est connue. Enfin, trois algorithmes décrits en FP sont analysés: la concaténation de deux listes, l'inversion double d'une liste de listes, et le tri par arbre binaire. Pour plus de détails sur cette approche, l'article de référence est [Cohen 88].

Nous nous plaçons au contraire dans le domaine des *séries génératrices*. Les combinatoriciens se servent de cet outil depuis longtemps, pour dénombrer des ensembles définis par des grammaires régulières et context-free. L'idée de l'étendre à l'analyse d'algorithmes en introduisant la notion de *descripteur de complexité* revient à P. Flajolet et J.M. Steyaert [FS 87].

1.5 Le système $\Lambda\Upsilon\Omega$

$\Lambda\Upsilon\Omega$ [FSZ] est un système d'aide à l'analyse d'algorithmes. Il est organisé suivant le schéma ci-dessus, adapté au domaine des séries génératrices:

1. le langage de description se nomme ADL (*Algorithm Description Language*)

2. le système d'équations a pour inconnues des séries génératrices et des descripteurs de complexité, qui sont des fonctions d'une variable complexe, soit z. Il est produit par le système ALAS.

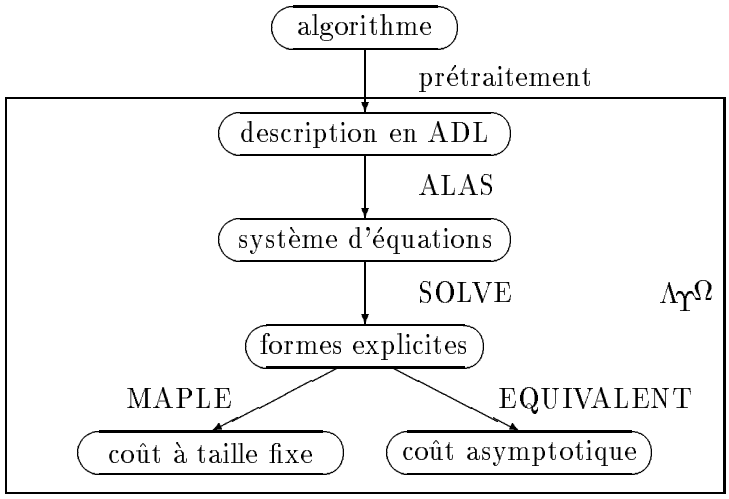
3. la phase dynamique consiste à résoudre ce système, c'est-à-dire à trouver une forme explicite pour chaque inconnue, à l'aide des fonctions usuelles (exp, log, ...) et d'opérateurs (de dérivation, d'intégration, ...). Le programme effectuant cette résolution s'appelle SOLVE.

4. l'analyse asymptotique donne le comportement des coefficients des séries lorsque leur rang tend vers l'infini. Plus précisément, si τ_f est le descripteur de complexité de la fonction f, et t la série génératrice des arguments de f, alors le coût moyen de f sur les données de taille n vaut:

$$\frac{[z^n]\tau_f}{[z^n]t}$$

   où $[z^n]g$ est le coefficient de z^n dans le développement de g. Le système d'analyse asymptotique (EQUIVALENT) est développé par B. Salvy à l'Ecole Polytechnique [Salvy 87].

5. le calcul du coût moyen pour une taille donnée k s'obtient en divisant $[z^k]\tau_f$ par $[z^k]t$. Pour cela, il suffit de disposer d'un système de calcul formel, par exemple MAPLE [MAPLE 85].


algorithme
prétraitement
description en ADL
ALAS
système d'équations
SOLVE
$\Lambda\Upsilon^{\Omega}$
formes explicites
MAPLE
EQUIVALENT
coût à taille fixe
coût asymptotique

Partie I

Les séries génératrices et l'analyse en moyenne d'algorithmes

Chapitre 2

Dénombrement de structures combinatoires

L'objet de ce chapitre est d'obtenir à partir de la définition d'un ensemble d'objets la série génératrice correspondante.

2.1 Définitions

2.1.1 Série génératrice ordinaire

Définition 1. *Soit $\mathcal{A}$ un ensemble d'objets, et une fonction taille, notée $|\ |$, de $\mathcal{A}$ dans* **N**. *La série génératrice ordinaire associée à $\mathcal{A}$ est:*

$$a(z) = \sum_{a \in \mathcal{A}} z^{|a|}$$

Par exemple, si $\mathcal{E} = \{\circ_1, \square_2\}$, $e(z) = z + z^2$ (les tailles des objets sont notées en indice).

2.1.2 Constructeur admissible

Un ensemble est défini:

1. soit par énumération (comme $\mathcal{E}$ ci-dessus),
2. soit à l'aide d'autres ensembles et d'un *constructeur*.

Ainsi, l'ensemble $\mathcal{F} = \mathcal{E}^*$ est formé à partir de $\mathcal{E}$ et du constructeur "séquence-de". $\mathcal{F}$ est l'ensemble des mots sur l'alphabet à deux lettres $\mathcal{E}$.

Un constructeur permet de former des structures compliquées à partir d'objets plus simples. Il en existe d'autres: l'union, le produit cartésien, le constructeur "ensemble de". Seuls nous intéressent les constructeurs dits *admissibles*:

Définition 2. *Soit $\mathcal{C} = \Phi(\mathcal{A}, \mathcal{B})$, où Φ est un constructeur, $\mathcal{A}$ et $\mathcal{B}$ des ensembles. Φ est dit admissible si les coefficients de $c(z)$ ne dépendent que de ceux de $a(z)$ et $b(z)$.*

Un constructeur admissible induit un opérateur (dont la forme peut être compliquée) sur les séries génératrices: $c(z) = \Theta(a, b)$.

2.1.3 Exemples

Le constructeur "union disjointe": $c_n = a_n + b_n$. Dans ce cas, l'opérateur est simplement la fonction somme sur les séries génératrices. Le constructeur "produit cartésien" est aussi admissible: $c_n = \sum_{k=0}^{n} a_k b_{n-k}$. L'opérateur associé est le produit de séries génératrices.

Il existe aussi des constructeurs qui ne sont pas admissibles. Le constructeur "intersection" en est un. En effet,

$$c_n = \mathrm{card}\{x \in \mathcal{A} \cap \mathcal{B}, |x| = n\}$$

ne peut pas s'exprimer uniquement avec a_n et de b_n. Il en est de même pour le constructeur "union". Pour qu'un constructeur ne soit pas admissible, il suffit qu'il fasse intervenir un caractère n'influant pas sur la taille. Par exemple, soit

$$\mathcal{E} = \{\clubsuit_1, \diamondsuit_1, \heartsuit_2, \spadesuit_3\}$$

le constructeur "est noir" n'est pas admissible: il n'existe pas de règle permettant de trouver le nombre d'objets noirs de taille n à partir de e_n.

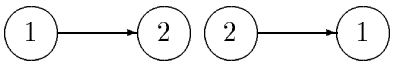
Un tableau de correspondance entre constructeurs admissibles et opérateurs sera donné au paragraphe 2.3.

2.2 Univers étiqueté et non étiqueté

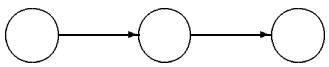
Il existe en réalité deux *univers*: l'un où les *atomes* sont indiscernables, l'autre où chaque *atome* porte une étiquette. Il faut voir les atomes comme les constituants de base des objets; un objet de taille n comporte n atomes. Dans l'*univers étiqueté*, chacun de ceux-ci porte un numéro (de 1 à n par exemple). Une des interprétations possibles de ces numéros est le classement des atomes par ordre d'ancienneté: celui qui porte le numéro 1 à été placé en premier lors de la construction de l'objet, etc ... Dans l'*univers non étiqueté*, les atomes ne sont pas numérotés.

2.2.1 Passage d'un univers à l'autre

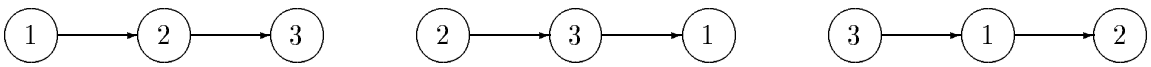
A tout objet étiqueté correspond un objet non étiqueté (son squelette). Celui-ci s'obtient en effaçant les numéros. Deux objets étiquetés sont dits identiques s'il est possible de les superposer (par "déformation continue") en faisant correspondre leurs indices. Ils ont nécessairement le même squelette. Mais des objets ayant le même squelette ne sont pas forcément identiques:

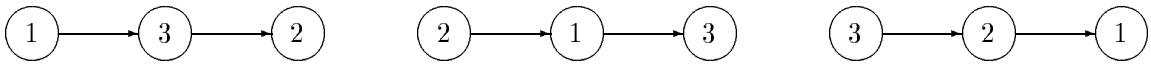


A l'inverse, à partir d'un objet non étiqueté, on forme un ou plusieurs objets étiquetés en répartissant des étiquettes. Par exemple, l'objet

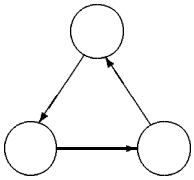


crée six objets étiquetés différents (c'est-à-dire non superposables):



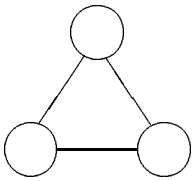


alors que

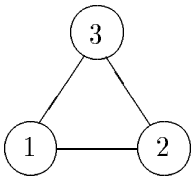


n'en crée que deux:

et que



n'en crée qu'un:



Ces exemples montrent qu'un objet non étiqueté de taille n, peut donner de 1 à $n!$ objets étiquetés, suivant sa structure.

2.2.2 Dénombrement d'objets étiquetés

Les séries génératrices ordinaires définies en 1 ne sont pas adaptées à un univers étiqueté. Elles conduisent à des opérateurs trop compliqués. La bonne définition est:

Définition 3. *La série génératrice exponentielle associée à un ensemble $\mathcal{A}$ est:*

$$\hat{a}(z) = \sum_{a \in \mathcal{A}} \frac{z^{|a|}}{|a|!}$$

Le nombre d'éléments de taille n d'un ensemble $\mathcal{A}$ est:

- a_n où a est la série génératrice ordinaire de $\mathcal{A}$
- $n!\,\hat{a}_n$ où $\hat{a}$ est la série génératrice exponentielle de $\mathcal{A}$.

2.2.3 La transformée de Laplace-Borel

La relation entre la série génératrice ordinaire et la série génératrice exponentielle d'un même ensemble est donc très simple: $a_n = n!\ \hat{a}_n$. L'opérateur correspondant est la transformée de Laplace-Borel:

$$a(t) = \int_0^\infty \hat{a}(tz) e^{-z} dz\,.$$

Démonstration:

$$\begin{aligned}\int_0^\infty \hat{a}(tz) e^{-z} dz &= \int_0^\infty \sum_{n=0}^\infty a_n \frac{(tz)^n}{n!} e^{-z} dz \\ &= \sum_{n=0}^\infty a_n \frac{t^n}{n!} \int_0^\infty z^n e^{-z} dz \\ &= \sum_{n=0}^\infty a_n t^n\end{aligned}$$

car $\int_0^\infty z^n e^{-z} dz$ vaut $n!$ (par récurrence et par parties). ∎

Il existe d'autres types de séries génératrices, notamment les séries génératrices de Dirichlet, qui sont utilisées pour compter des paramètres multiplicatifs (par opposition à la taille qui est un paramètre additif). Nous n'en parlerons pas plus ici.

2.3 Constructeurs et opérateurs

Ce paragraphe énumère pour chaque type d'univers, les constructeurs admissibles les plus courants, et l'opérateur qu'ils induisent sur les séries génératrices.

2.3.1 Univers non étiqueté

union disjointe	$\mathcal{A} = \mathcal{B} + \mathcal{C}$	$\Longrightarrow$	$a(z) = b(z) + c(z)$
produit cartésien	$\mathcal{A} = \mathcal{B} \times \mathcal{C}$	$\Longrightarrow$	$a(z) = b(z)\, c(z)$
séquence	$\mathcal{A} = \mathcal{B}^*$	$\Longrightarrow$	$a(z) = \dfrac{1}{1 - b(z)}$
ensemble	$\mathcal{A} = 2^{\mathcal{B}}$	$\Longrightarrow$	$a(z) = \exp(\Phi(b))$
multi-ensemble	$\mathcal{A} = \mathcal{M}(\mathcal{B})$	$\Longrightarrow$	$a(z) = \exp(\Psi(b))$

Φ est l'opérateur de Pólya:

$$\Phi(b) = \frac{b(z)}{1} - \frac{b(z^2)}{2} + \frac{b(z^3)}{3} - \cdots$$

et Ψ est défini par:

$$\Psi(b) = \frac{b(z)}{1} + \frac{b(z^2)}{2} + \frac{b(z^3)}{3} + \cdots$$

Origine de l'opérateur de Pólya: lorsque $\mathcal{A}$ est l'ensemble des parties de $\mathcal{B}$, on peut écrire

$$\mathcal{A} = \prod_{b \in \mathcal{B}} (\epsilon \cup b) \tag{2.1}$$

(développer mentalement pour vérifier l'exactitude de cette équation), ce qui se traduit pour les séries génératrices en:

$$\begin{aligned} a(z) &= \prod_{b\in\mathcal{B}}(1+z^{|b|}) \\ &= \prod_{n=0}^{\infty}(1+z^n)^{b_n} \end{aligned} \tag{2.2}$$

puis en passant au logarithme:

$$\begin{aligned} \log a(z) &= \sum_{n=0}^{\infty} b_n \log(1+z^n) \\ &= \sum_{n=0}^{\infty} b_n (z^n - z^{2n}/2 + z^{3n}/3 - \cdots) \\ &= \sum_{n=0}^{\infty} b_n z^n - 1/2 \sum_{n=0}^{\infty} b_n z^{2n} + 1/3 \sum_{n=0}^{\infty} b_n z^{3n} - \cdots \\ &= b(z) - 1/2 b(z^2) + 1/3 b(z^3) - \cdots \end{aligned} \tag{2.3}$$

Pour le constructeur multi-ensemble, il suffit de remplacer $\epsilon \cup b$ par b^* dans l'égalité 2.1. Dans l'équation 2.2, $1+z^{|b|}$ devient $1/(1-z^{|b|})$, et dans la formule finale 2.3, les signes $-$ deviennent des signes $+$.

2.3.2 Univers étiqueté

Certains des constructeurs ci-dessus n'ont pas d'équivalent lorsque les objets portent des numéros. Ainsi, la distinction entre ensemble et multi-ensemble n'existe plus dans l'univers étiqueté. Pour plus de détail sur la notion de produit partitionnel (ou *shuffle*), et pour une définition des constructeurs ci-dessous, une bonne référence est [Vitter 87].

union disjointe	$\mathcal{A} = \mathcal{B} + \mathcal{C}$	$\Longrightarrow$	$\hat{a}(z) = \hat{b}(z) + \hat{c}(z)$
produit partitionnel	$\mathcal{A} = \mathcal{B} \times \mathcal{C}$	$\Longrightarrow$	$\hat{a}(z) = \hat{b}(z)\ \hat{c}(z)$
complexe partitionnel	$\mathcal{A} = \mathcal{B}^{<*>}$	$\Longrightarrow$	$\hat{a}(z) = \dfrac{1}{1-\hat{b}(z)}$
complexe partitionnel abélien	$\mathcal{A} = \mathcal{B}^{[*]}$	$\Longrightarrow$	$\hat{a}(z) = \exp(\hat{b})$
cycle	$\mathcal{A} = \mathcal{C}(\mathcal{B})$	$\Longrightarrow$	$\hat{a}(z) = \log \frac{1}{1-\hat{b}(z)}$
cycle non orienté	$\mathcal{A} = \mathcal{UC}(\mathcal{B})$	$\Longrightarrow$	$\hat{a}(z) = \frac{1}{2}\log \frac{1}{1-\hat{b}(z)} + \frac{\hat{b}(z)}{2} + \frac{\hat{b}^2(z)}{4}$

Chapitre 3

Descripteurs de complexité

Ce chapitre introduit les concepts de descripteur de complexité et de schéma admissible, puis montre comment les utiliser pour calculer le coût moyen d'un algorithme.

3.1 Définitions

3.1.1 Mesure additive

Une mesure (de complexité) est dite *additive* si la complexité d'une suite d'instructions est la somme des complexités des instructions composantes. Par exemple, le temps d'exécution, le nombre de passages en un certain point sont des mesures additives. Par contre, la place mémoire n'en est pas une. Nous nous intéressons ici uniquement à des mesures additives. Le terme de "coût" sera utilisé dans ce sens.

3.1.2 Programme, procédure et fonction

Un *programme* consiste en une suite d'une ou de plusieurs instructions, qui sont exécutées séquentiellement. Une *procédure* est une partie de programme à laquelle on a donné un nom (souvent en vue d'une utilisation multiple), et qui ne renvoie aucun résultat. Si $\mathcal{A}$ est l'ensemble des arguments possibles d'une procédure P, on note $P : \mathcal{A}$. Une *fonction* est une procédure qui retourne un objet. Si $\mathcal{A}$ est l'ensemble des arguments possibles d'une fonction Q, et si $\mathcal{B}$ contient l'ensemble des valeurs retournées, on note $Q : \mathcal{A} \to \mathcal{B}$.

3.1.3 Descripteur de complexité

Définition 4. *Soit un programme P, dont l'ensemble des données est $\mathcal{A}$. Le coût de l'exécution de P avec une donnée a est noté $\mu P(a)$. Le* descripteur de complexité ordinaire *associé à P est:*

$$\tau P(z) = \sum_{a \in \mathcal{A}} \mu P(a)\, z^{|a|}$$

Le coefficient de z^n dans $\tau P(z)$ est la complexité *cumulée* de P sur les objets de taille n. Lorsque ces objets sont équiprobables, la complexité *moyenne* de P (sur les objets de taille n) est donc:

$$\frac{[z^n]\tau P(z)}{[z^n]a(z)}$$

où $a(z)$ est la série génératrice ordinaire associée à $\mathcal{A}$. Le descripteur de complexité exponentiel s'obtient en remplaçant $z^{|a|}$ par $z^{|a|}/|a|!$ (au contraire, cette définition prend en compte des modèles uniformes avec *équiprobabilité* de toutes les entrées de même taille).

EXEMPLE: Si P a un coût constant μ pour tout $a \in \mathcal{A}$, le descripteur de complexité ordinaire est simplement le produit de μ par la série génératrice ordinaire de $\mathcal{A}$:

$$\tau P(z) = \mu.a(z)$$

3.1.4 Schéma admissible

Un *schéma* de programmation sert à assembler un ou plusieurs programmes pour en construire un plus complexe. C'est l'analogue d'un constructeur pour les classes de structures: soient deux programmes P et Q, d'ensembles de données respectifs $\mathcal{A}$ et $\mathcal{B}$. Un nouveau programme $R = \Gamma(P,Q)$ est construit à partir de P, Q et d'un schéma binaire Γ. Soit $\mathcal{C}$ l'ensemble des données de R. Notre but est de calculer le descripteur de complexité de R à partir de ceux de P et Q, d'où la notion d'admissibilité:

Définition 5. *Un schéma Γ est dit admissible si:*

1. *il existe un constructeur admissible Φ tel que $\mathcal{C} = \Phi(\mathcal{A},\mathcal{B})$*
2. *les coefficients de τR ne dépendent que de ceux de τP, τQ, a et b.*

Dans ce cas, on note Λ l'opérateur tel que $\tau R = \Lambda(\tau P, \tau Q, a, b)$.

3.2 Schémas indépendants des constructeurs

NOTATIONS:

- $P : \mathcal{A}$ signifie que P est une procédure dont l'ensemble des données est $\mathcal{A}$,
- $Q : \mathcal{A} \to \mathcal{B}$ signifie que Q est une fonction dont l'ensemble des données est $\mathcal{A}$, et qui renvoie un élément de $\mathcal{B}$.

3.2.1 Séquence d'instructions

Lorsque P et Q sont deux programmes de même ensemble de données $\mathcal{A}$ (ce qui sera noté désormais $P : \mathcal{A}, Q : \mathcal{A}$), un nouveau programme R est formé en exécutant d'abord P, puis Q, ce qui se note $R \stackrel{\text{def}}{=} P;Q$. Pour $a \in \mathcal{A}$, le coût de $R(a)$ vaut:

$$\mu R(a) = \mu P(a) + \mu Q(a). \tag{3.1}$$

Le schéma "exécution séquentielle" est donc admissible:

Règle 1 (Exécution séquentielle)

$$P : \mathcal{A}, \quad Q : \mathcal{A}, \quad R \stackrel{\text{def}}{=} P;Q \quad \Longrightarrow$$

$$\tau R = \tau P + \tau Q$$

3.2.2 La conditionnelle if ... then ... else

Le schéma en question est:

$$P : \mathcal{A}, \quad Q : \mathcal{A}, \quad R(a) \stackrel{\text{def}}{=} \textbf{if } T(a) \textbf{ then } P(a) \textbf{ else } Q(a) \tag{3.2}$$

T est une fonction booléenne, renvoyant *true* ou $false$.

Règle 2 (Sélection des tailles) *Le schéma 3.2 est admissible lorsque T teste uniquement les tailles: $T(a) = true \iff |a| \in K$, où $K \subset \mathbf{N}$:*

$$P : \mathcal{A}, \quad R(a) \stackrel{\text{def}}{=} \textbf{if } T(a) \textbf{ then } P(a) \textbf{ else } Q(a) \quad \Longrightarrow$$

$$\tau R(z) = \tau T(z) + \sum_{n \in K} \tau P_n z^n + \sum_{n \in \mathbf{N}-K} \tau Q_n z^n$$

Démonstration: soit $\mathcal{D} = \{a \in \mathcal{A} \mid T(a) = true\}$:

$$\mu R(a) = \mu T(a) + \begin{cases} \mu P(a) & \text{si } a \in \mathcal{D} \\ \mu Q(a) & \text{sinon} \end{cases}$$

d'où en sommant séparément sur chaque $\mathcal{D}_n$:

$$\tau R(z) = \tau T(z) + \sum_{n \geq 0} \Big(\sum_{a \in \mathcal{D}_n} \mu P(a) \Big) z^n + \sum_{n \geq 0} \Big(\sum_{a \notin \mathcal{D}_n} \mu Q(a) \Big) z^n \tag{3.3}$$

Le terme $S_n = \sum_{a \in \mathcal{D}_n} \mu P(a)$ est le coût total de P sur une partie de $\mathcal{A}_n$, en l'occurrence $\mathcal{D}_n$. La seule série génératrice pouvant nous informer sur le coût de P est $\tau P(z)$. Or $\tau P(z)$ ne contient que le coût *total* sur $\mathcal{A}_n$. Lorsque $\mathcal{D}_n$ est une partie propre de $\mathcal{A}_n$, $\tau P(z)$ ne suffit donc pas pour calculer S_n. Sinon, deux cas sont possibles:

1. $\mathcal{D}_n = \emptyset$: $S_n = 0$
2. $\mathcal{D}_n = \mathcal{A}_n$: $S_n = \tau P_n$.

Soit alors $K = \{n \in \mathbf{N} \mid \mathcal{D}_n = \mathcal{A}_n\}$. L'équation 3.3 donne exactement la formule donnée par la règle ci-dessus. ∎

EXEMPLES:

- test d'infériorité:

$$R(c) = \textbf{ if } size(c) \leq m \textbf{ then } P(c) \textbf{ else } Q(c) \quad \Longrightarrow$$

$$\tau R(z) = \tau_{size}(z) + \sum_{n \leq m} \tau P_n z^n + \sum_{n > m} \tau Q_n z^n$$

- test d'égalité:

$$R(c) = \textbf{ if } size(c) = m \textbf{ then } P(c) \textbf{ else } Q(c) \quad \Longrightarrow$$

$$\tau R(z) = \tau_{size}(z) + \tau P_m z^m + \sum_{n \neq m} \tau Q_n z^n$$

- test de congruence:

$$R(c) = \textbf{ if } size(c) \bmod j = m \textbf{ then } P(c) \textbf{ else } Q(c) \quad \Longrightarrow$$

$$\tau R(z) = \tau_{size}(z) + \sum_{n \bmod j = m} \tau P_n z^n + \sum_{n \bmod j \neq m} \tau Q_n z^n$$

- test de parité (cas particulier de test de congruence pour $j = 2$):

$$R(c) = \textbf{ if } size(c) \bmod 2 = 0 \textbf{ then } P(c) \textbf{ else } Q(c) \quad \Longrightarrow$$

$$\tau R(z) = \tau_{size}(z) + \frac{1}{2}[\tau P(z) + \tau P(-z)] + \frac{1}{2}[\tau Q(z) - \tau Q(-z)]$$

Nous avons étudié ci-dessus le cas où P et R ont même ensemble de définition. Voyons maintenant le cas où $P : \mathcal{A}$ et $R : \mathcal{A} \cup \mathcal{B}$:

Règle 3 (Sélection du type)

$$P : \mathcal{A}, \quad R : \mathcal{A} \cup \mathcal{B}, \quad R(c) \stackrel{\text{def}}{=} \textbf{ if } c \in \mathcal{A} \textbf{ then } P(c) \quad \Longrightarrow$$

$$\tau R(z) = \tau P(z)$$

Démonstration: laissée au lecteur.

REMARQUE: la composition n'est pas admissible !

Le schéma de composition est:

$$P : \mathcal{B}, \quad Q : \mathcal{A} \to \mathcal{B}, \quad R : \mathcal{A}, \quad R(a) = P(Q(a))$$

(si $P : \mathcal{B} \to \mathcal{C}$ est une fonction, alors $R : \mathcal{A} \to \mathcal{C}$). Pour $a \in \mathcal{A}$,

$$\mu R(a) = \mu Q(a) + \mu P(Q(a))$$

ce qui donne

$$\tau R(z) = \sum_{a \in \mathcal{A}} \mu Q(a) z^{|a|} + \sum_{a \in \mathcal{A}} \mu P(Q(a)) z^{|a|}$$

La première somme se calcule sans problème: c'est $\tau Q(z)$. Quant à la seconde, son coefficient de z^n vaut

$$\sum_{a \in \mathcal{A}_n} \mu P(Q(a))$$

La distribution des tailles de $Q(a)$ est quelconque en général. Il n'y a donc aucune chance de pouvoir exprimer cette somme à l'aide des τP_n. Même s'il existait une relation:

$$|a| = n \Longrightarrow |Q(a)| = f(n)$$

cela ne suffirait pas, car la distribution de $Q(a)$ sur $\mathcal{B}_{f(n)}$ est également quelconque: un objet peut avoir plusieurs antécédents alors qu'un autre n'en a pas. Or le descripteur de complexité τP nous permet de totaliser le coût de P lorsqu'il y a une distribution uniforme sur chaque $\mathcal{B}_n$ (cela signifie que chaque $b \in \mathcal{B}_n$ est atteint le même nombre de fois par Q).

3.3 Schémas de descente dans les constructeurs

Les algorithmes opérant sur une structure complexe S exécutent souvent des opérations indépendantes sur des composantes de S. Supposons que l'on ait défini un type *répertoire* qui soit une séquence de fiches. Une fiche contient un nom, un prénom et un numéro de téléphone. Nous serons amenés à effectuer des opérations typiques telles que:

1. la descente dans une composante fixée (prendre la 3ème fiche, lire le prénom d'une fiche),
2. la descente dans une composante aléatoire (prendre une fiche au hasard),
3. la descente dans toutes les composantes (prendre toutes les fiches, lire toutes les informations d'une fiche).

Un autre exemple est la recherche d'une occurrence déterminée (1) ou quelconque (2) d'un motif dans une expression, ou de toutes ses occurrences (3). Chacun de ces trois types de descente est étudié dans un des paragraphes ci-dessous.

3.3.1 Descente dans une composante déterminée

Il s'agit des schémas:

$$P : \mathcal{A}, \quad R : \Phi(\mathcal{A}), \quad R(c) \stackrel{\text{def}}{=} P(a_j)$$

où j est fixé. Par convention, si c contient moins de j composantes, aucune opération n'est effectuée. Sinon, le programme P est exécuté sur la j-ème composante de c. Cela n'a de sens que si ces composantes sont ordonnées, c'est-à-dire si Φ est le produit cartésien ou la séquence dans un univers non-étiqueté, le produit partitionnel ou le complexe partitionnel dans un univers étiqueté.

Univers non étiqueté

Constructeur produit cartésien Comme il n'y a que deux composantes, seuls les choix $j = 1$ et $j = 2$ donnent un coût non nul.

Règle 4 (Descente dans la première composante d'un produit)

$$P : \mathcal{A}, \quad R : \mathcal{A} \times \mathcal{B}, \quad R((a, b)) \stackrel{\text{def}}{=} P(a) \quad \Longrightarrow$$

$$\tau R(z) = \tau P(z) b(z)$$

Démonstration:

$$\begin{aligned} \tau R(z) &= \sum_{a \in \mathcal{A}, b \in \mathcal{B}} \mu P(a) z^{|a|+|b|} \\ &= \sum_{a \in \mathcal{A}} \mu P(a) z^{|a|} \sum_{b \in \mathcal{B}} z^{|b|} \\ &= \tau P(z) b(z) \ \blacksquare \end{aligned}$$

Pour ce qui est de la descente dans la 2ème composante, il suffit de remplacer $b(z)$ par $a(z)$ dans la règle 4.

Constructeur séquence Soit k le nombre d'éléments de la séquence. Si $j > k$ (ce qui inclut le cas $k = 0$ de la séquence vide), aucune opération n'est effectuée.

Règle 5 (Descente dans la j-ième composante d'une séquence)

$$P : \mathcal{A}, \quad R : \mathcal{A}^*, \quad R((a_1, \ldots, a_k)) \stackrel{\text{def}}{=} \textbf{ if } k \geq j \textbf{ then } P(a_j) \quad \Longrightarrow$$

$$\tau R(z) = \tau P(z) \frac{a(z)^{j-1}}{1 - a(z)}$$

Démonstration:

$$\begin{aligned} \tau R(z) &= \sum_{k \geq j} \sum_{(a_1, \ldots, a_k) \in \mathcal{A}^k} \mu P(a_j) z^{|a_1| + \cdots + |a_k|} \\ &= \sum_{k \geq j} \sum_{a_j} \mu P(a_j) z^{|a_j|} \sum_{a_1, \ldots, a_{j-1}, a_{j+1}, \ldots, a_k} z^{|a_1| + \cdots + |a_{j-1}| + |a_{j+1}| + \cdots + |a_k|} \\ &= \sum_{k \geq j} \tau P(z) a(z)^{k-1} \quad \blacksquare \end{aligned}$$

Univers étiqueté

Les résultats sont analogues au cas non étiqueté.

Constructeur produit partitionnel

Règle 6 (Descente dans la première composante d'un produit partitionnel)

$$P : \mathcal{A}, \quad R : \mathcal{A} \times_P \mathcal{B}, \quad R((a, b)) \stackrel{\text{def}}{=} P(a) \quad \Longrightarrow$$

$$\hat{\tau} R(z) = \hat{\tau} P(z) \hat{b}(z)$$

Démonstration:

$$\hat{\tau} R(z) = \sum_{a,b} \binom{|a| + |b|}{|a| \ |b|} \mu P(a) \frac{z^{|a| + |b|}}{(|a| + |b|)!} = \hat{\tau} P(z) \hat{b}(z) \quad \blacksquare$$

Constructeur complexe partitionnel

Règle 7 (Descente dans la j-ième composante d'un complexe partitionnel)

$$P : \mathcal{A}, \quad R : \mathcal{A}^{<*>}, \quad R((a_1, \ldots, a_k)) \stackrel{\text{def}}{=} \textbf{ if } k \geq j \textbf{ then } P(a_j) \quad \Longrightarrow$$

$$\hat{\tau} R(z) = \hat{\tau} P(z) \frac{\hat{a}(z)^{j-1}}{1 - \hat{a}(z)}$$

3.3.2 Descente dans une composante aléatoire

Il s'agit des schémas:

$$P : \mathcal{A}, \quad R : \Phi(\mathcal{A}), \quad R(c) \stackrel{\text{def}}{=} P(a_{random(k)})$$

où k est le nombre de composantes de c, et $random(k)$ est un entier entre 1 et k, tiré selon une loi uniforme. Il faut noter ici que toutes les composantes doivent être de même type.

Univers non étiqueté

Les résultats de ce paragraphe pour les constructeurs ensemble et multi-ensemble ne sont pas simples. Cela est dû au fait que ces constructeurs n'ont pas de décomposition naturelle en univers non-étiqueté. D'ailleurs, les opérateurs qu'ils induisaient sur les séries génératrices n'étaient pas simples non plus.

Constructeur produit cartésien

Règle 8 (Descente dans un produit)

$$P : \mathcal{A}, \quad R : \mathcal{A} \times \mathcal{A}, \quad R((a_1, a_2)) \stackrel{\text{def}}{=} P(a_{random(2)}) \quad \Longrightarrow$$
$$\tau R(z) = \tau P(z) a(z)$$

Démonstration: appliquer la règle 4 à chacun des deux choix, et prendre la moyenne.

Constructeur séquence

Règle 9 (Descente dans une séquence)

$$P : \mathcal{A}, \quad R : \mathcal{A}^*, \quad R((a_1, \ldots, a_k)) \stackrel{\text{def}}{=} P(a_{random(k)}) \quad \Longrightarrow$$
$$\tau R(z) = \tau P(z) \frac{1}{1 - a(z)}$$

Démonstration: d'après la remarque ci-dessus, il est équivalent de choisir une composante au hasard ou de prendre toujours la première. Le résultat est donc un cas particulier de la règle 5, avec $j = 1$. ∎

Constructeur ensemble

Règle 10 (Descente dans un ensemble)

$$P : \mathcal{A}, \quad R : 2^{\mathcal{A}}, \quad R(\{a_1, \ldots, a_k\}) \stackrel{\text{def}}{=} P(a_{random(k)}) \quad \Longrightarrow$$
$$\tau R(z) = \sum_{n \geq 1} \mu P_n z^n \left[\int \frac{c(z,u)\, du}{1 + uz^n} \right]_{u=1}$$

où $c(z,u) = \prod_{a \in \mathcal{A}} (1 + uz^{|a|})$ *(*z *compte la taille des ensembles et* u *leur cardinal).*

Démonstration: soit $\tau R^k(z)$ le descripteur de complexité de R sur les ensembles de k éléments $c = \{a_1, \ldots, a_k\}$. Pour $k = 0$, $\tau R^k(z) = 0$. Pour $k \geq 1$, le coût total pour $j = random(k)$ est égal à la moyenne du coût sur les k valeurs possibles de j:

$$\begin{aligned}
k\ \tau R^k(z) &= \sum_{j=1}^{k} \sum_{\{a_1,\ldots,a_k\} \subset 2^{\mathcal{A}}} \mu P(a_j) z^{|a_1|+\cdots+|a_k|} \\
&= \Big(\sum_{\{a_1,\ldots,a_k\} \subset 2^{\mathcal{A}}} \mu P(a_1) z^{|a_1|+\cdots+|a_k|} + \cdots \Big) \\
&= k \sum_{\{a_1,\ldots,a_k\} \subset 2^{\mathcal{A}}} \mu P(a_1) z^{|a_1|+\cdots+|a_k|} \\
&= \sum_{a_1} \mu P(a_1) z^{|a_1|} \sum_{\{a_2,\ldots,a_k\} \subset 2^{\mathcal{A}-\{a_1\}}} z^{|a_2|+\cdots+|a_k|} \qquad (3.4)
\end{aligned}$$

En utilisant le fait que

$$\sum_{\{b_1,\ldots,b_k\}\subset 2^{\mathcal{B}}} z^{|b_1|+\cdots+|b_k|} = [u^k]\prod_{b\in\mathcal{B}}(1+uz^{|b|})$$

la dernière sommation de 3.4 s'écrit:

$$[u^{k-1}]\frac{\prod_{a\in\mathcal{A}}(1+uz^{|a|})}{1+uz^{|a_1|}}$$

et ne dépend plus que de $|a_1|$. Rassemblons donc les termes correspondant à $|a_1| = n$:

$$\tau R^k(z) = \frac{1}{k}\sum_n \mu P_n z^n [u^{k-1}]\frac{c(z,u)}{1+uz^n} \tag{3.5}$$

Nous pouvons à présent sommer par rapport à k:

$$\tau R(z) = \sum_{n\geq 1}\mu P_n z^n \left[\int \frac{c(z,u)\,du}{1+uz^n}\right]_{u=1} \quad \blacksquare$$

Constructeur multi-ensemble
Règle 11 (Descente dans un multi-ensemble)

$$P:\mathcal{A},\quad R:\mathcal{M}(\mathcal{A}),\quad R(\{a_1,\ldots,a_k\}) \stackrel{\text{def}}{=} P(a_{random(k)}) \quad \Longrightarrow$$

$$\tau R(z) = \sum_{n\geq 1}\mu P_n z^n \left[\int \frac{c(z,u)\,du}{1-uz^n}\right]_{u=1}$$

où $c(z,u) = \prod_{a\in\mathcal{A}}\frac{1}{1-uz^{|a|}}$.

Démonstration: pour un multi-ensemble $\{a_1,\ldots,a_k\}$, la contribution μP de chaque élément est à multiplier par $1/k$ pour obtenir la moyenne. Sommons par rapport à l'élément a sur lequel P est appelé, et par rapport au nombre k d'éléments distincts de a:

$$\begin{aligned}
\tau R(z) &= \sum_{a\in\mathcal{A}}\mu P(a)\sum_{k\geq 0}\left([u^k]\prod_{a'\in\mathcal{A}-\{a\}}\frac{1}{1-uz^{|a'|}}\left(\frac{z^{|a|}}{k+1}+\frac{2z^{2|a|}}{k+2}+\cdots\right)\right)\\
&= \sum_{a\in\mathcal{A}}\mu P(a)\sum_{k\geq 0}\left([u^k](1-uz^{|a|})c(z,u)\left(\frac{z^{|a|}}{k+1}+\frac{2z^{2|a|}}{k+2}+\cdots\right)\right)\\
&= \sum_{a\in\mathcal{A}}\mu P(a)\left(\sum_{k\geq 0}[u^k]c(z,u)\left(\frac{z^{|a|}}{k+1}+\frac{2z^{2|a|}}{k+2}+\cdots\right)-\sum_{k\geq 0}[u^k]c(z,u)\left(\frac{z^{2|a|}}{k+1}+\frac{2z^{3|a|}}{k+2}+\cdots\right)\right)\\
&= \sum_{a\in\mathcal{A}}\mu P(a)\sum_{k\geq 0}[u^k]c(z,u)\left(\frac{z^{|a|}}{k+1}+\frac{z^{2|a|}}{k+2}+\cdots\right)\\
&= \sum_{a\in\mathcal{A}}\mu P(a)\sum_{k\geq 0}\left([u^k]c(z,u)\frac{z^{|a|}}{k+1}+[u^{k+1}]c(z,u)\frac{uz^{2|a|}}{k+2}+\cdots\right)\\
&= \sum_{a\in\mathcal{A}}\mu P(a)\sum_{k\geq 0}\frac{1}{k+1}[u^k]c(z,u)\frac{z^{|a|}}{1-uz^{|a|}}\\
&= \sum_{a\in\mathcal{A}}\mu P(a)z^{|a|}\sum_{k\geq 0}\frac{1}{k+1}[u^k]\frac{c(z,u)}{1-uz^{|a|}}
\end{aligned}$$

Enfin, on rassemble les termes correspondant à des éléments de même taille:

$$\tau R(z) = \sum_{n\geq 1} \mu P_n z^n \left[\int \frac{c(z,u)\,du}{1-uz^n} \right]_{u=1} \blacksquare$$

Univers étiqueté

Pour le produit et le complexe partitionnel, les résultats découlent de ceux obtenus avec un choix déterministe, comme en univers non étiqueté.

Constructeur produit partitionnel

Règle 12 (Descente dans un produit partitionnel)

$$P : \mathcal{A}, \quad R : \mathcal{A} \times_P \mathcal{A}, \quad R((a_1, a_2)) \stackrel{\text{def}}{=} P(a_{random(2)}) \quad \Longrightarrow$$

$$\hat{\tau} R(z) = \hat{\tau} P(z) \hat{a}(z)$$

Constructeur complexe partitionnel

Règle 13 (Descente dans un complexe partitionnel)

$$P : \mathcal{A}, \quad R : \mathcal{A}^{<*>}, \quad R((a_1, \ldots, a_k)) \stackrel{\text{def}}{=} P(a_{random(k)}) \quad \Longrightarrow$$

$$\hat{\tau} R(z) = \hat{\tau} P(z) \frac{1}{1-\hat{a}(z)}$$

Constructeur complexe partitionnel abélien

Règle 14 (Descente dans un complexe partitionnel abélien)

$$P : \mathcal{A}, \quad R : \mathcal{A}^{[*]}, \quad R(\{a_1, \ldots, a_k\}) \stackrel{\text{def}}{=} P(a_{random(k)}) \quad \Longrightarrow$$

$$\hat{\tau} R(z) = \frac{\exp(\hat{a}(z)) - 1}{\hat{a}(z)} \hat{\tau} P(z)$$

Démonstration: comme pour le constructeur ensemble, il faut calculer séparément $\hat{\tau} R^k(z)$:

$$\hat{\tau} R^k(z) = \sum_{\{a_1, \ldots, a_k\}} \mu P(a_{random(k)}) \frac{z^{|a_1|+\cdots+|a_k|}}{(|a_1| + \cdots + |a_k|)!}$$

Or le nombre d'ensembles de k composantes du complexe partitionnel abélien (cpa) est $1/k!$ fois celui du complexe partitionnel (cp), d'où:

$$\begin{aligned} \hat{\tau} R^k_{cpa}(z) &= \frac{1}{k!} \hat{\tau} R^k_{cp}(z) \\ &= \frac{1}{k!} \hat{\tau} P(z) \hat{a}^{k-1}(z) \end{aligned} \tag{3.6}$$

La sommation donne bien la formule 14. $\blacksquare$

Constructeur cycle

Règle 15 (Descente dans un cycle)

$$P : \mathcal{A}, \quad R : \mathcal{C}(\mathcal{A}), \quad R([a_1, \ldots, a_k]) \stackrel{\text{def}}{=} P(a_{random(k)}) \quad \Longrightarrow$$

$$\hat{\tau} R(z) = \frac{\hat{\tau} P(z)}{\hat{a}(z)} \log \frac{1}{1 - \hat{a}(z)}$$

Démonstration: le nombre de cycles de k composantes est $1/k$ fois le nombre d'éléments du complexe partitionnel, d'où:

$$\begin{aligned} \hat{\tau} R^k_{cycle}(z) &= \frac{1}{k} \hat{\tau} R^k_{cp}(z) \\ &= \frac{1}{k} \hat{\tau} P(z) \hat{a}^{k-1}(z) \end{aligned} \tag{3.7}$$

ce qui conduit à la formule 15. ■

3.3.3 Schémas de descente dans toutes les composantes

Nous étudions ici le cas où R appelle P sur *toutes* les composantes de la structure c. Lorsque le nombre de composantes est fixe (produit), il suffit d'appliquer les résultats du paragraphe précédent. Cependant, pour les constructeurs à nombre de composantes variable, comme la séquence ou l'ensemble, les résultats ne sont pas immédiats, quoique en général plus simples que ceux ci-dessus.

Univers non étiqueté

Constructeur séquence

Règle 16 (Descente totale dans une séquence)

$$P : \mathcal{A}, \quad R : \mathcal{A}^*, \quad R((a_1, \ldots, a_k)) \stackrel{\text{def}}{=} P(a_1); \ldots; P(a_k) \quad \Longrightarrow$$

$$\tau R(z) = \frac{\tau P(z)}{(1 - a(z))^2}$$

Démonstration: d'après la règle 4, il vient:

$$\tau R^k(z) = k \tau P(z) a^{k-1}(z)$$

d'où le résultat. ■

Constructeur ensemble

Règle 17 (Descente totale dans un ensemble)

$$P : \mathcal{A}, \quad R : 2^{\mathcal{A}}, \quad R(\{a_1, \ldots, a_k\}) \stackrel{\text{def}}{=} P(a_1); \ldots; P(a_k) \quad \Longrightarrow$$

$$\tau R(z) = c(z)(\tau P(z) - \tau P(z^2) + \tau P(z^3) - \cdots)$$

où $c(z) = \exp(\Phi(a))$.

Démonstration: en reprenant la valeur de $\tau R^k(z)$ calculée en 3.5, et en la multipliant par k (puisque P est appelé pour toutes les composantes), il vient:

$$\tau R^k(z) = \sum_n \mu P_n z^n [u^{k-1}] \frac{c(z,u)}{1+uz^n}$$

ce qui donne après sommation:

$$\begin{aligned}
\tau R(z) &= \sum_n \mu P_n z^n \sum_{k\geq 1} [u^{k-1}] \frac{c(z,u)}{1+uz^n} \\
&= \sum_n \mu P_n z^n \frac{c(z)}{1+z^n} \\
&= c(z) \sum_n \mu P_n z^n (1 - z^n + z^{2n} - \cdots) \\
&= c(z)(\tau P(z) - \tau P(z^2) + \tau P(z^3) - \cdots). \blacksquare
\end{aligned}$$

Constructeur multi-ensemble

Règle 18 (Descente totale dans un multi-ensemble)

$$P : \mathcal{A}, \quad R : \mathcal{M}(\mathcal{A}), \quad R(\{a_1, \ldots, a_k\}) \stackrel{\text{def}}{=} P(a_1); \ldots; P(a_k) \quad \Longrightarrow$$

$$\tau R(z) = c(z)(\tau P(z) + \tau P(z^2) + \tau P(z^3) + \cdots)$$

où $c(z) = \exp(\Psi(a))$.

Univers étiqueté

Constructeur complexe partitionnel

Règle 19 (Descente totale dans un complexe partitionnel)

$$P : \mathcal{A}, \quad R : \mathcal{A}^{<*>}, \quad R((a_1, \ldots, a_k)) \stackrel{\text{def}}{=} P(a_1); \ldots; P(a_k) \quad \Longrightarrow$$

$$\hat{\tau} R(z) = \frac{\hat{\tau} P(z)}{(1 - \hat{a}(z))^2}$$

Démonstration: analogue à celle concernant la séquence.

Constructeur complexe partitionnel abélien

Règle 20 (Descente totale dans complexe partitionnel abélien)

$$P : \mathcal{A}, \quad R : \mathcal{A}^{[*]}, \quad R(\{a_1, \ldots, a_k\}) \stackrel{\text{def}}{=} P(a_1); \ldots; P(a_k) \quad \Longrightarrow$$

$$\hat{\tau} R(z) = \exp(\hat{a}(z)) \hat{\tau} P(z)$$

Démonstration: multiplions par k la valeur de $\hat{\tau} R^k$ déterminée en 3.6:

$$\hat{\tau} R^k(z) = \frac{1}{(k-1)!} \hat{\tau} P(z) \hat{a}^{k-1}(z)$$

d'où la formule ci-dessus. $\blacksquare$

Constructeur cycle

Règle 21 (Descente totale dans un cycle)

$$P : \mathcal{A}, \quad R : cycle(\mathcal{A}), \quad R([a_1, \ldots, a_k]) \stackrel{\text{def}}{=} P(a_1); \ldots; P(a_k) \quad \Longrightarrow$$

$$\hat{\tau} R(z) = \frac{\hat{\tau} P(z)}{1 - \hat{a}(z)}$$

Démonstration: multiplions par k la valeur de $\hat{\tau} R^k$ déterminée en 3.7:

$$\hat{\tau} R^k(z) = \hat{\tau} P(z) \hat{a}^{k-1}(z)$$

ce qui, après sommation sur k, donne bien le résultat annoncé. ∎

3.4 Tableaux récapitulatifs

univers non étiqueté	$\tau R(z)$
$P : \mathcal{A}, \quad Q : \mathcal{A}, \quad R \stackrel{\text{def}}{=} P; Q$	$\tau P(z) + \tau Q(z)$
$P : \mathcal{A}, \quad R(a) \stackrel{\text{def}}{=}$ **if** $T(a)$ **then** $P(a)$ **else** $Q(a)$	$\tau T(z) + \sum_{n \in K} \tau P_n z^n + \sum_{n \in \mathbf{N} - K} \tau Q_n z^n$
$T(a) \Longleftrightarrow \lvert a \rvert \in K$	
$R(c) =$ **if** $size(c) \leq m$ **then** $P(c)$ **else** $Q(c)$	$\tau_{size}(z) + \sum_{n \leq m} \tau P_n z^n + \sum_{n > m} \tau Q_n z^n$
$R(c) =$ **if** $size(c) = m$ **then** $P(c)$ **else** $Q(c)$	$\tau_{size}(z) + \tau P_m z^m + \sum_{n \neq m} \tau Q_n z^n$
$R(c) =$ **if** $size(c) \bmod j = m$ **then** $P(c)$ **else** $Q(c)$	$\tau_{size}(z) + \sum_{n \bmod j = m} \tau P_n z^n + \sum_{n \bmod j \neq m} \tau Q_n z^n$
$R(c) =$ **if** $size(c) \bmod 2 = 0$ **then** $P(c)$ **else** $Q(c)$	$\tau_{size}(z) + \frac{1}{2}[\tau P(z) + \tau P(-z)] + \frac{1}{2}[\tau Q(z) - \tau Q(-z)]$
$P : \mathcal{A}, \quad R : \mathcal{A} \cup \mathcal{B}, \quad R(c) \stackrel{\text{def}}{=}$ **if** $c \in \mathcal{A}$ **then** $P(c)$	$\tau P(z)$
$P : \mathcal{A}, \quad R : \mathcal{A} \times \mathcal{B}, \quad R((a, b)) \stackrel{\text{def}}{=} P(a)$	$\tau P(z) b(z)$
$P : \mathcal{A}, \quad R : \mathcal{A}^*, \quad R((a_1, \ldots, a_k)) \stackrel{\text{def}}{=}$ **if** $k \geq j$ **then** $P(a_j)$	$\tau P(z) \frac{a(z)^{j-1}}{1 - a(z)}$
$P : \mathcal{A}, \quad R : \mathcal{A} \times \mathcal{A}, \quad R((a_1, a_2)) \stackrel{\text{def}}{=} P(a_{random(2)})$	$\tau P(z) a(z)$
$P : \mathcal{A}, \quad R : \mathcal{A}^*, \quad R((a_1, \ldots, a_k)) \stackrel{\text{def}}{=} P(a_{random(k)})$	$\tau P(z) \frac{1}{1 - a(z)}$
$P : \mathcal{A}, \quad R : 2^{\mathcal{A}}, \quad R(\{a_1, \ldots, a_k\}) \stackrel{\text{def}}{=} P(a_{random(k)})$	$\sum_{n \geq 1} \mu P_n z^n \left[\int \frac{c(z,u)\, du}{1 + u z^n} \right]_{u=1}$
$P : \mathcal{A}, \quad R : \mathcal{M}(\mathcal{A}), \quad R(\{a_1, \ldots, a_k\}) \stackrel{\text{def}}{=} P(a_{random(k)})$	$\sum_{n \geq 1} \mu P_n z^n \left[\int \frac{c(z,u)\, du}{1 - u z^n} \right]_{u=1}$

univers étiqueté	$\hat{\tau} R(z)$
$P : \mathcal{A}, \quad R : \mathcal{A} \times_P \mathcal{B}, \quad R((a, b)) \stackrel{\text{def}}{=} P(a)$	$\hat{\tau} P(z) \hat{b}(z)$
$P : \mathcal{A}, \quad R : \mathcal{A}^{<*>}, \quad R((a_1, \ldots, a_k)) \stackrel{\text{def}}{=}$ **if** $k \geq j$ **then** $P(a_j)$	$\hat{\tau} P(z) \frac{\hat{a}(z)^{j-1}}{1 - \hat{a}(z)}$
$P : \mathcal{A}, \quad R : \mathcal{A} \times_P \mathcal{A}, \quad R((a_1, a_2)) \stackrel{\text{def}}{=} P(a_{random(2)})$	$\hat{\tau} P(z) \hat{a}(z)$
$P : \mathcal{A}, \quad R : \mathcal{A}^{<*>}, \quad R((a_1, \ldots, a_k)) \stackrel{\text{def}}{=} P(a_{random(k)})$	$\hat{\tau} P(z) \frac{1}{1 - \hat{a}(z)}$

Les règles 10, 11, 14, 17, 18, 20 et 21 sont des résultats nouveaux. Une question se pose maintenant: comment utiliser ces règles pour analyser tout un programme ? C'est l'objet du chapitre suivant.

Chapitre 4

Applications

Ce chapitre montre d'abord comment utiliser les résultats du chapitre précédent pour analyser toute une procédure (et non plus seulement une seule instruction); puis on définit la classe des programmes que l'on peut analyser à l'aide des descripteurs de complexité.

4.1 Analyse d'une procédure

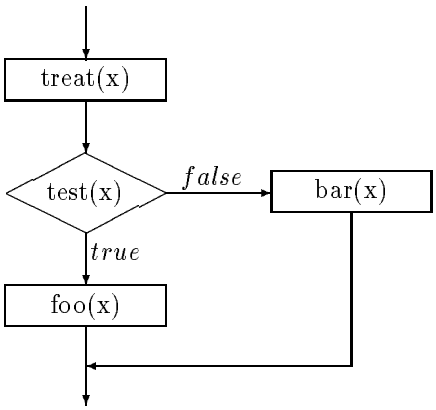
Le corps d'une procédure est un graphe orienté, ayant une entrée et une seule sortie. Ce graphe contient deux types de noeuds:

- les noeuds *séquentiels*,
- les noeuds-*test*.

Par exemple, au programme

```
treat(x);
if test(x) = true
then foo(x)
else bar(x);
```

est associé le graphe



Les noeuds séquentiels ont un seul successeur, alors que les noeuds-test en ont deux (*true* et *false*). A une exécution de la procédure sur une donnée correspond un parcours du graphe. Le coût de cette exécution est la somme des coûts des instructions se trouvant sur le parcours, pour la donnée en question. Parmi ces instructions, nous distinguons:

1. les instructions *prédéfinies*, dont le coût ne dépend pas de la donnée (l'addition de deux entiers par exemple),
2. les instructions *définies par l'utilisateur* (par une procédure ou une fonction).

Les instructions définies par l'utilisateur opèrent sur des objets (ou types) qu'il a construit lui-même, pouvant donc avoir des tailles quelconques (listes, arbres, graphes). Leur coût dépend en général de la taille de ces objets. Par conséquent, il est plus difficile à déterminer que celui des instructions prédéfinies.

4.1.1 Contribution d'une instruction prédéfinie

Soit une procédure $P : \mathcal{A}$ dont le corps contient l'instruction prédéfinie $I(x)$. Soit $\mathcal{A}'$ l'ensemble des $a \in \mathcal{A}$ pour lequel le graphe de $P(a)$ passe par I. La contribution de $I(x)$ à τP est:

$$\sum_{a \in \mathcal{A}'} \mu I(x) z^{|a|}$$

ou encore puisque μI est constant:

$$\mu I \sum_{a \in \mathcal{A}'} z^{|a|} = \mu I . a'(z)$$

Il suffit donc de connaître $a'(z)$. Cela signifie que pour chaque noeud-test, l'ensemble des objets qui partent dans la branche *true* (respectivement *false*) est *combinatoirement énumérable.*

Définition 6. *Un ensemble d'objets est* combinatoirement énumérable *s'il est possible de le décrire de manière* finie *à l'aide de constructeurs et d'ensembles énumérés.*

Par exemple, tout ensemble fini est combinatoirement énumérable (il suffit de l'énumérer).

4.1.2 Contribution d'un appel de procédure

Soit une procédure $P : \mathcal{A}$ dont le corps contient l'appel $Q(x)$, où $Q : \mathcal{B}$ est une procédure définie par l'utilisateur. On définit $\mathcal{A}'$ comme précédemment. La contribution de $Q(x)$ à τP est:

$$S = \sum_{a \in \mathcal{A}'} \mu Q(x) z^{|a|} \tag{4.1}$$

Ici, x dépend de a (par exemple la branche gauche d'un arbre binaire, le troisième élément d'une séquence, un élément quelconque d'un cycle, ...).

Cas d'une distribution régulière

En sommant par rapport à la taille de $x(a)$, 4.1 devient:

$$S = \sum_n \sum_{|b|=n} \mu Q(b) \sum_{x(a)=b} z^{|a|}$$

Hypothèse 1 [Régularité à taille fixée]: $\sum_{x(a)=b} z^{|a|}$ ne dépend que de $|b| = n$. Lorsque cette hypothèse est vérifiée, S devient:

$$\begin{aligned} S &= \sum_n \left(\sum_{|b|=n} \mu Q(b) \right) \left(\sum_{|x(a)|=n} z^{|a|} \right) \\ &= \sum_n \mu Q_n \sum_{|x(a)|=n} z^{|a|} \\ &= \sum_n \mu Q_n z^n \sum_{|x(a)|=n} z^{|a|-n} \end{aligned}$$

HYPOTHÈSE 2 [RÉGULARITÉ GLOBALE]: $\sum_{|x(a)|=n} z^{|a|-n}$ ne dépend pas de n. Cela signifie qu'il n'y a pas de corrélation entre $|x|$ et la taille du reste de a. C'est le cas par exemple lorsque a est le produit de x et d'une autre partie y: la somme en question est alors la série génératrice associée à y. Lorsque les hypothèses 1 et 2 sont vérifiées, la distribution $x(a), a \in \mathcal{A}$ est dite *régulière* dans $\mathcal{B}$. Dans ce cas, la contribution de $Q(x)$ s'exprime à l'aide de τQ et de la série génératrice f, indépendante de n, de l'hypothèse 2:

$$S = \tau Q(z) f(z)$$

Encore faut-il pouvoir calculer f. A nouveau, ceci est possible lorsque $\mathcal{A}$ est un produit cartésien.

Autres cas

Lorsque la distribution n'est pas régulière dans $\mathcal{B}$, la contribution ne peut pas s'exprimer en fonction de τQ. Il faut alors remplacer l'instruction $Q(x)$ par le corps de la procédure Q. Ce processus s'arrête si l'on ne tombe pas dans une chaîne infinie de distributions non régulières.

Exemple de chaîne finie: la dérivation formelle d'une expression s'écrit:

```
case e of
 expo(e1)          : times(diff(e1),copy(e));
 ...
```

où e_1 est régulière, mais pas e. Il faut donc remplacer $copy(e)$ par:

```
case e of
 expo(e1)          : expo(copy(e1));
 ...
```

et le processus s'arrête car e_1 est régulière.

Exemple de chaîne infinie: la fonction *distrib* qui développe $(v + w) \times u$ en $v \times u + w \times u$ s'écrit:

```
function distrib(e:expression):expression;
case e of
 times(t,u):
 case t of
 plus(v,w)         : plus(distrib(times(v,u)),distrib(times(w,u)));
 times(v,w)        : times(distrib(t),distrib(u));
 expo(v)           : times(expo(distrib(v)),distrib(u));
 x                 : times(x,distrib(u))
 end
 ...
end;
```

Les appels $distrib(times(v, u))$ et $distrib(times(w, u))$ conduisent à une chaîne infinie correspondant à un arbre dont la branche gauche ne contient que des +. Un moyen de contourner ce problème est de définir le type $expression1 = times(expression, expression)$ (cf section 10.2.9), et d'introduire une nouvelle fonction, $distrib1$, qui opère sur ce type:

```
function distrib(e:expression):expression;
case e of
 plus(t,u)         : plus(distrib(t),distrib(u));
 times(t,u)        : distrib1(e);
 expo(t)           : expo(distrib(t));
 x                 : x
end;

function distrib1(e:expression1):expression;
case e of
 times(t,u):
```

```
case t of
plus(v,w)          : plus(distrib1(times(v,u)),
                      distrib1(times(w,u)));
times(v,w): times(distrib1(t),distrib(u));
expo(v)            : times(expo(distrib(v)),distrib(u));
x                  : times(x,distrib(u))
end
end;
```

A présent, tous les appels de procédure sont réguliers. Ce travail de régularisation, qui est fait à la main dans la version actuelle de $\Lambda\Upsilon^{\Omega}$, peut s'automatiser, à condition que le programme ne teste qu'une partie finie des structures (le symbole *times* ci-dessus). De plus, cette partie doit être située au sommet de la structure (à la racine pour un arbre), de telle sorte que les sous-structures soient régulières (e_1 ci-dessus).

Théorème 1. *Un appel du type $Q(x)$ dans le corps d'une procédure $P : \mathcal{A}$, où $Q : \mathcal{B}$ est définie par l'utilisateur, est analysable si l'ensemble $\{x(a) \mid a \in \mathcal{A}\}$ est combinatoirement énumérable et si:*

1. *soit $x(a)$ est régulière dans $\mathcal{B}$,*
2. *soit l'expansion de Q conduit à un processus fini.*

De plus, le cas 2 se ramène par régularisation au cas 1.

4.2 Classe des programmes analysables

Un programme est analysable lorsque toutes les procédures ou fonctions qui y sont définies le sont. Une procédure est analysable lorsque:

1. pour chaque instruction prédéfinie, l'ensemble des objets pour lesquels elle est exécutée est combinatoirement énumérable,
2. chaque appel de procédure ou de fonction est analysable (cf ci-dessus).

Il faut remarquer qu'un appel nécessite de connaître une description de la distribution $x(a)$ (à l'aide de constructeurs), alors qu'il suffit de la série génératrice pour une instruction prédéfinie.

Part II

The algebraic analyzer

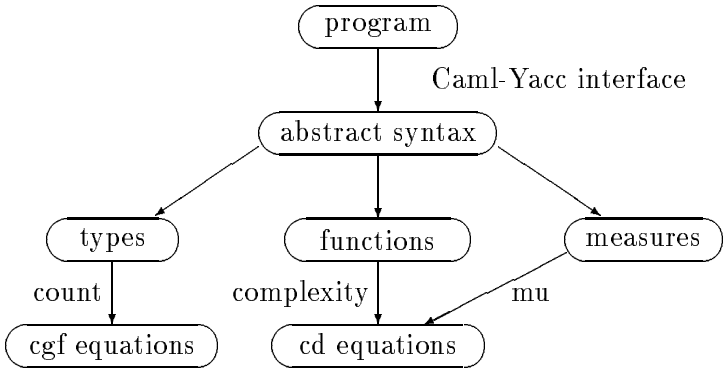
Chapter 5

The Caml implementation

The first implementation of an algebraic analyzer was in `LeLisp`, a dialect of `Lisp` developped at Inria. It was only dealing with simple families of trees. So types declarations were very restricted. Nevertheless, it could analyze non trivial algorithms such as formal differentiation . This chapter describes the current implementation, in the Caml programming language(Categorical Abstract Machine Language), developped by the Formel project at Inria (Rocquencourt, France) [CAM].

5.1 Principle

The following diagram describes how we translate a program into equations for counting generating functions and complexity descriptors.



5.2 From file to syntax tree

This phase is very easy in Caml, which has a Yacc interface. All you have to do is:

- write the syntax rules (those of the preceeding chapter).
- write an action *in Caml* for each rule (usually the creation of an object).
- define types for these objects.

For the non-terminal `instruction`, rules and actions are:

```
instruction : CASE ident_list OF cases END    {Cases($2,$4)}
            | BEGIN instructions END          {Instructions($2)}
            | IF condition THEN instruction
              ELSE instruction                {IfThenElse($2,$4,$6)}
            | FORALL IDENT IN IDENT
                instruction                   {Forall($2,$4,$5)}
            | expression                      {Expression($1)}
            |                                 {Empty_instr}
            ;
```

where the tokens `CAML`, `OF`, `END`, `BEGIN`, `IF`, `THEN`, `ELSE`, `FORALL` and `IN` stand respectively for `caml`, `of`, `end`, `begin`, `if`, `then`, `else`, `forall` and `in`. The *predefined* token `IDENT` stands for any sequence of letters or digits, beginning with a letter. In the next version will be provided an instruction to descent in one random component (`forone`).
The Caml type declaration of `instruction` is:

```
type instruction = Cases of string list & (pattern list & instruction) list
                 | Instructions of instruction list
                 | IfThenElse of condition & instruction & instruction
                 | Forall of string & string & instruction
                 | Expression of expression
                 | Empty_instr ;;
```

So the Adl phrase (in *concrete syntax*)

begin
if a = b **then** nil
else case a **of**
c : nil;
otherwise : count
end
end;

will be automatically parsed into a Caml object (or *abstract syntax*):

```
(Instructions
 [(IfThenElse
   ((Equal ((Ident "a"),(Pat_id "b"))),(Expression (Exp_id "nil")),
    (Cases
     (["a"],
      [([(Pat_id "c")],(Expression (Exp_id "nil")));
       ([Pat_all],(Expression (Exp_id "count")))])))))])],
```

All other functions will deal with such well-typed objects.

5.3 From type to counting generating function

5.3.1 Choice of a generating function type

The first step is the choice of the type of generating functions. In the current implementation, labelled and unlabelled atoms can not coexist. So this determination is very easy:

- if there are only unlabelled atoms, choose an ordinary generating function
- if there are only labelled atoms, choose an exponential generating function
- if there are both or none, send an error message

This type will stand for of all generating functions, counting generating functions as well as complexity descriptors.

5.3.2 Counting

This step is simply an application of formulae of section 2.3.

5.4 From function to complexity descriptor

5.4.1 Need of a grammar specification

If P is a program, whose data are taken in $\mathcal{A}$, our purpose is to find an equation for the total cost of P on all data, which is given by:

$$\tau_P(z) = \sum_{a \in \mathcal{A}} \mu_P(a) z^{|a|}$$

An *elementary instruction* is an instruction that can not be decomposed into several ones (it is the equivalent of an atom for objects). In a program, elementary instructions are of two types:

1. instructions of constant cost
2. instructions whose cost depends on one (or more) variables

Addition of two integers is an instruction of the first type. Its total cost is of the form:

$$\mu \sum_{a \in \mathcal{A}_1} z^{|a|}$$

where μ is the instruction cost measure (for example time for an addition), and $\mathcal{A}_1$ is the subset of elements of $\mathcal{A}$ for which this instruction is executed. Hence it suffices to know the generating function of $\mathcal{A}_1$. Instructions of the second type are essentially procedure calls. For example, if procedure Q is called on variable b, the total cost is:

$$\sum_{a \in \mathcal{A}_1} \mu_Q(b) z^{|a|}$$

where again $\mathcal{A}_1$ is the subset of elements of $\mathcal{A}$ for which this instruction is executed. Of course, b depends on a. In the case where $\mathcal{A}_1 = \mathcal{B} \times \mathcal{C}$, we can rewrite the above expression:

$$\left(\sum_{b \in \mathcal{B}} \mu_Q(b) z^{|b|} \right) \left(\sum_{c \in \mathcal{C}} z^{|c|} \right)$$

Hence we only need to know $c(z)$, the generating function of $\mathcal{C}$, and the total cost of Q on data in $\mathcal{B}$. If the data set of Q is exactly $\mathcal{B}$, this total cost is in fact $\tau_Q(z)$, which is given by another equation. We need to know if a given set $\mathcal{B}$ is equal to the data set $\mathcal{D}$ of a procedure Q. In Q, we have a grammar specification of $\mathcal{D}$. Hence we need a grammar specification for $\mathcal{B}$. We call such a specification a *type description*. Operations we must be able to do on types are:

- comparison
- intersection with a pattern
- subtraction

Comparison is needed when we encounter a procedure call $P(b)$, where $P : \mathcal{A}$, $b \in \mathcal{B}$ and $\mathcal{B} \subset \mathcal{A}$. If $\mathcal{B} = \mathcal{A}$, the total cost is simply τP. Hence we must be able to decide whether two type descriptions are equal or not. Intersection with a pattern is needed for instructions like:

$$\textbf{case } a \textbf{ of } p1 : exp1; \ldots \textbf{ end}$$

We must find which sub-type $\mathcal{A}'$ of $\mathcal{A}$ matches pattern $p1$. And then, we must subtract $\mathcal{A}'$ to $\mathcal{A}$, which gives us a type description for others patterns. Caml functions for this three operations are **eq_type** for comparison, **filter** for intersection with a pattern, and **subtract** for subtraction. There are five functions to compute a procedure complexity:

1. **compl_call** for a procedure or function call
2. **compl_expr** for an expression
3. **compl** for an instruction
4. **compl_cases** for the special instruction **case**
5. **complexity** takes a function name and computes its complexity descriptor.

Chapter 6

Maple Interface

6.1 The maplecaml command

When we have to do some symbolic manipulations (simplifications, factorizations), why would we write routines that already exist ? The Maple language does such computations. To run Maple and Caml simultaneously, just execute:

```
% maplecaml
```

and you will start an usual Caml session, with Caml welcome message. Then if you type:

```
lib_load "maple";;
```

Maple welcome message will be displayed on the standard output. When you leave Caml session by

```
quit();;
```

Maple will be killed automatically.

6.2 Sending a command to Maple ...

♣ `maple_eval : (string -> void)`

This command puts a command into the Maple input buffer, and displays the Maple response on standard output.

6.3 ...and receiving a result from Maple

♣ `to_maple : (string -> string)`

This command differs from `maple_eval` for two reasons:

1. there is no echo on the standard output,
2. it returns a string containing the result of Maple last evaluation (this is done by the Maple function `lprint`).

6.4 A Maple toplevel

♣ `maple : (void -> void)`

When you type:

```
maple();;
```

you switch from Caml's toplevel to Maple's one. This command is useful when you have to give several commands to Maple, and you do not want to type each time `maple_eval` or `to_maple`. To return to Caml, just type "quit".

6.5 Maple errors

When an error occurs during evaluation of a Maple phrase, the corresponding message (of Maple) is displayed, and the exception failure is raised, with value "maple":

```
#maple_eval "a:=b + - 1;";;
syntax error before: - 1;

Evaluation Failed: maple
```

6.6 An example

This is a script of a MapleCaml session:

```
% maplecaml 35
   CAML (sun) (V 2-5) by INRIA Fri Jan 22

#lib_load "maple";;
Initializing maple ...
    |\^/|
._|\|   |/|_. INRIA - Rocquencourt
 \  MAPLE  /  Version 4.2 --- Dec 1987
 <____ ____>  For on-line help, type  help();
      |
#let binomial n k = maple_eval("binomial("^string_of_num n^","^
# string_of_num k^");");;
Value binomial = - : (num -> num -> void)

#binomial 10 5;;
                                      252

() : void

#let binomial1 n k = num_of_string(to_maple("binomial("^string_of_num n^","^
```

```
# string_of_num k^");"));;
Value binomial1 = - : (num -> num -> num)

#binomial1 100 49;;
98913082887808032681188722800 : num

#maple_eval "a:=1;";;
                                        a := 1

() : void

#maple_eval "b:=2;";;
                                        b := 2

() : void

#maple();;
> a;
                                          1

> b;
                                          2

> a+b;
                                          3

> quit
() : void

#quit();;
A bientot ...
%
```

Chapter 7

Solve: the dynamic phase

Alas produces equations for counting generating functions and complexity descriptors. In current implantation stage, the analytic analyzer Ananas [Salvy 87] works on generating functions given by an explicit form. Hence we need a program to obtain these explicit forms from Alas equations. This Maple program, Solve, will certainly be a part of Ananas in the future.

7.1 Principle

Equations for counting generating functions do not depend on complexity descriptors, hence we can solve them first.

7.1.1 Counting generating functions

When no recursivity is used in type definitions, the system is triangular. To obtain explicit forms, we just assign each right member to the corresponding variable. When recursivity is used, we encounter *fixed point equations*. A typical form is

$$f = P(f)$$

where P is a polynom, whose coefficients are integers, powers of z or other generating functions. Let $d = degree(P)$. When $d = 1$, the solution is a rational expression in z and other generating functions. When $d = 2$, we obtain a second degree equation, hence two solutions. Which one is the good one ? We remark that f is analytic at $z = 0$, so $\lim_{z \to 0} f(z)$ is finite. This suffices to distinguish the right solution. For higher degree ($d \geq 3$), explicit solutions are very large, and the basic operations achieved by the analytic analyzer (mainly comparison or simplification) may become very expensive.

7.1.2 Complexity descriptors

As above, non-recursive functions give non-recursive equations, and recursive functions give recursive equations for complexity descriptors. But in this last case, equations are always linear in terms

of complexity descriptors. Hence we get a linear system:

$$\begin{pmatrix} \tau_1 \\ \vdots \\ \tau_n \end{pmatrix} = \begin{pmatrix} a_{11} & \dots & a_{1n} \\ \vdots & & \vdots \\ a_{n1} & \dots & a_{nn} \end{pmatrix} \begin{pmatrix} \tau_1 \\ \vdots \\ \tau_n \end{pmatrix} + \begin{pmatrix} b_1 \\ \vdots \\ b_n \end{pmatrix}$$

where the a_{ij} and b_i are generating functions in z.

7.2 Implementation in Maple

This section only describes some functions. A complete listing of Solve is given in appendix B. The main function, `Solve`, takes two sets of equations:

```
Solve := proc(type_eqs,compl_eqs)
local oldRootOf,sol1,sol2;

    sol1 := my_solve(type_eqs);

    oldRootOf := op(RootOf);
    RootOf := op(myRootOf);

    sol1 := eval(sol1);

    RootOf := op(oldRootOf);

    sol2 := my_solve(compl_eqs);

    sol1 union sol2;
end:
```

where `my_solve` solves a system with respect to variables appearing in left members of it:

```
my_solve := proc(s) solve(s,my_indets(s)) end:

my_indets := proc(l) map(proc(x) op(1,x) end,l) end:
```

The two lines

```
    oldRootOf := op(RootOf);
    RootOf := op(myRootOf);
```

are a way of redefining Maple's function `RootOf`, which is used in a system for equations up from degree 2. The new function `myRootOf` will detect equations of degree 2 and solve them:

```
myRootOf := proc(eqn)
local sol,sol1,i,d,l,var;

   var := map(proc(x) if whattype(x)=`string` then x fi end,
```

```
      indets(eqn) minus {_Z});
   if nops(var)<>1 then ERROR(`bad number of variables`,var) fi;
   var := var[1];
   d := degree(eqn,_Z);
   if d=2 then
      sol := solve(eqn,_Z);
      sol1 := {};
      for i to d do
         l := limit(sol[i],var=0);
         if l <> `undefined`
         then sol1 := sol1 union {sol[i]};
         fi;
      od;
      if nops(sol1)=1 then RETURN(sol1[1]);
      else RETURN('RootOf(eqn)');
      fi;
   else RETURN('RootOf(eqn)');
   fi;
end:
```

and

```
    RootOf := op(oldRootOf);
```

resets the initial value of `RootOf`. `Solve` returns the union of solutions for counting generating functions and for complexity descriptors. To assign them, just type:

```
assign(Solve({...},{...}));
```

Part III

$\Lambda\Upsilon^{\Omega}$: Reference Manual